ESTRUTURAS DE DADOS E ALGORITMOS
Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 6
INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

6.1. Objetivos

Objetivos que o aluno deve atingir com esta ficha:
- Compreender o conceito de classe e objeto;
- Compreender o conceito de construtor e destrutor;
- Aplicar exemplos de ponteiros para classes.

6.2. Classes: declaração e implementação

A programação orientada a objetos tem como principais objetivos reduzir a complexidade no desenvolvimento de software e aumentar a sua produtividade.

Um objeto é uma abstração de software que pretende representar um determinado tipo de dados (informação) e também o conjunto de operações ou funções de manipulação desses dados. Um objeto é formado por um conjunto de atributos (variáveis) e métodos (procedimentos, funções ou rotinas). Os atributos possuem um tipo de dados que define os possíveis valores que o atributo pode representar, como um número inteiro, um número real ou uma *string*. Os métodos são rotinas que, quando executadas, realizam alguma tarefa específica, como alterar o valor de um atributo do objeto.

Uma classe é uma descrição ou um modelo para um objeto. Uma classe define de forma abstrata os atributos e os métodos que representam características de um conjunto de objetos semelhantes. O conjunto de todos os atributos e métodos de uma classe designa-se por membros da classe.

O conceito de classe é um dos pilares da programação orientada a objetos, por permitir a reutilização efetiva de código. Um objeto é definido como uma instância (ou concretização) de uma determinada classe. Para melhor compreender as vantagens da utilização de objetos em programação, tomemos o seguinte exemplo:

```
// Estrutura de dados que define um retângulo
typedef struct {
   // Atributos do Retângulo
   int altura;
   int comprimento;
} TRetangulo;
```

Se pretendêssemos implementar uma série de funções para manipular retângulos definidos através desta estrutura, ter-se-ia de criar funções com um parâmetro deste tipo, como a função que se apresenta a seguir:

```
// Desenha um retângulo com as caraterísticas guardadas numa variável
// do tipo estrutura TRetangulo
void desenhaRetangulo(TRetangulo &ret) {

   // O que será que fazem as linhas que se seguem?
   cout << endl;
   for (int i=0; i < ret.altura; i++) {
      for (int j=0; j < ret.comprimento; j++) {
          if((i>0) && (i<ret.altura-1) && (j>0) && (j<ret.comprimento-1))
             cout << " ";
          else
             cout << "*";
      }
      cout << endl; // Para que serve esta instrução?
   }
   cout << endl;
}
//------------------------------------------------------------------------------
int main() {
   TRetangulo umRetangulo;

   umRetangulo.altura  = 5;
   umRetangulo.comprimento   = 7;

   desenhaRetangulo(umRetangulo);
}
```

O C++ permite resolver o mesmo problema de uma forma muito mais expedita, utilizando uma classe de objetos para representar um retângulo. Neste caso, seria possível combinar os atributos do retângulo e a função que o desenha numa *única* ***declaração*** e ***definição*** (ou ***implementação***). A declaração da classe de objetos do tipo retângulo poderia ser algo semelhante ao seguinte:

```
// Classe de retângulos
class CRetangulo {
  private:
     // Atributos do retângulo
     int altura;
     int comprimento;

  public:
     // Método que desenha o retângulo
     void desenha(void);
}; // O «;» é essencial: não esquecer ou ocorrerão erros de
   //  compilação noutras partes do programa !!!
```

A implementação do método `desenha()` definido no âmbito da classe `CRetangulo` poderá ser a seguinte:

```
// Definição/implementação do método que desenha o retângulo
// guardado pela classe
// Veja-se o nome da classe seguida do "scope resolution operator" ::
// é essencial para o compilador saber a que classe pertence o método...
void CRetangulo::desenha(void) {
   for (int i=0; i < altura; i++) {
      for (int j=0; j < comprimento; j++) {
          if ((i>0) && (i<altura-1) && (j>0) && (j<comprimento-1))
             cout << " ";
          else
             cout << "*";
      }
      cout << endl;
   }
   cout << endl;
}
```

Notas:

- O nome de uma classe obedece normalmente a uma convenção que determina que este seja precedido por um prefixo C (a indicar a definição de uma classe de objetos).
- As palavras reservadas `private`, `public` e `protected` determinam/controlam a forma como é permitido o acesso aos membros da classe.
  - `private`: Os membros privados de uma classe só são acessíveis a partir de métodos internos da classe ou então por objetos de classes declaradas como «amigas».
  - `protected`: Os membros protegidos de uma classe são visíveis a partir de métodos internos da classe, por objetos de classes declaradas como «amigas», e também por objetos de todas as suas classes derivadas.
  - `public`: Os membros públicos de uma classe são visíveis em qualquer ponto onde a classe seja visível.
  - Se declararmos membros de uma classe sem especificarmos qualquer tipo de visibilidade, então esses membros são por omissão do tipo `private`.

6.3. Objetos: construtores e destrutores, instanciação e membros estáticos

Uma classe não faz sentido se não houver maneira de a instanciar. Assim, analogamente aos tipos normais que se instanciam através de variáveis, os **objetos** permitem instanciar as classes. Desta forma, num programa em C++ existirão quase sempre vários objetos da mesma classe, cada um correspondendo a uma instanciação diferente, ocupando regiões de memória diferentes e representando, provavelmente, dados diferentes.

Em cada classe existe um método obrigatório que convém ser declarado e definido e que determina a forma como se podem fazer instanciações de uma classe: o **construtor por omissão ou por defeito**. Este método declara-se sempre da mesma forma: tem o nome da classe, não tem argumentos (parâmetros) e é

sempre `public`. Na sua definição, é esperado que se inicializem os atributos da classe, mas não é obrigatório fazê-lo. O construtor é *sempre* chamado automaticamente quando um novo objeto é criado (instanciado).

O código definido a seguir complementa a classe anterior definindo um construtor por omissão:

```
...
// Classe de retângulos
class CRetangulo {
  private:
     // atributos do retângulo
     int altura;
     int comprimento;

  public:
     // Construtor por defeito ou por omissão
     CRetangulo() {
          altura = comprimento = 0;
     }
     // Método que desenha o retângulo
     void desenha(void);
};
```

Como se pode observar, foi definido um construtor por defeito que permite inicializar os atributos do retângulo a zero. Neste exemplo, podemos também verificar que é possível definir métodos dentro da declaração da classe: chama-se a isto **definição *inline***. Normalmente, apenas se utiliza esta forma de definir os métodos em situações em que estes não são extensos, para não prejudicar a legibilidade do código e em particular da definição da classe.

O C++ permite também definir membros de uma classe que são partilhados por todos os objetos dessa classe: os chamados membros estáticos ou `static`. Estes membros ficam disponíveis para todos os objetos, uma espécie de variável global da classe. Uma utilidade para este tipo de métodos é apresentada no exemplo que se segue: imagine-se que se quer contabilizar o número de objetos de uma classe; não existe maneira de o fazer automaticamente, a não ser recorrendo a um membro `static` que guarde essa informação. Para manter a boa prática de encapsulamento, pode-se usar um membro `static` mas `private` e aceder a ele a partir de métodos. Vejamos um exemplo utilizando mais uma vez a classe `CRetangulo`:

```
...
// Classe de retângulos
class CRetangulo {
  private:
     // Atributos do retângulo
     int altura;
     int comprimento;

     // Membro acessível a todos os objetos desta classe,
     // ocupando apenas uma zona de memória
     static int numRet;
```

```
public:
   // Construtor por omissão ou defeito
   CRetangulo() {
        altura = comprimento = 0;
        numRet++;
   }

   // Construtor com parâmetros
   CRetangulo(int a, int c) {
       altura = a;
        comprimento = c;
        numRet++;
   }

  // Construtor por cópia
   CRetangulo(const CRetangulo &r) {
       altura = r.altura;
        comprimento = r.comprimento;
        numRet++;
   }

   // Destrutor
   ~CRetangulo() {
        numRet--;
   }

   // Método que inicializa os atributos do retângulo
   void inicializaTamanho(int a, int c) {
     altura = a;
     comprimento = c;
   }

   // Método que devolve os atributos do retângulo (por referência)
   void devolveTamanho(int &a, int &c) {
     a = altura;
     c = comprimento;
   }

   // Método que utiliza o atributo "static" para imprimir o número
   // total de objetos deste tipo.
   void imprimeNumRetangulos(void){
     cout << "O numero atual de objetos-retângulo e': " << numRet;
     cout << endl;
   }


   // Método que utiliza o atributo "static" para devolver o número
   // total de objetos deste tipo
```

```
      int devolveNumRetangulos(void){
        // A utilização do ponteiro "this" era desnecessária, mas mostra
        // como funciona...
        return this->numRet;
      }

      // Método que desenha o retângulo
      void desenha(void);

};  // fim da classe CRetangulo
//-----------------------------------------------------------------
int CRetangulo::numRet = 0;  // Definição do membro estático
//-----------------------------------------------------------------
...
```

O programa principal que usaria a classe poderia ser o seguinte:

```
...
int main(void) {
   // Cria retângulo e chama construtor por omissão
   CRetangulo retangulo;

   // Cria retângulo utilizando construtor por enumeração
   CRetangulo outroRetangulo (7, 5);

   // Desenha o retângulo
   outroRetangulo.desenha();

   retangulo.inicializaTamanho(5, 7);

   // Desenha outro retângulo
   retangulo.desenha();

   // Cria outro objeto retângulo e usa a sobrecarga alternativa
   // para o construtor
   CRetangulo ultimoRetangulo(4, 5);

   // Imprime o número de objetos-retângulo criados até ao momento
   ultimoRetangulo.imprimeNumRetangulos();

   // Criação de uma tabela estática de objetos-retângulo
   CRetangulo variosRets[5];


   // Criação de uma tabela dinâmica de objetos-retângulo
   CRetangulo *eMaisEstesRets = new CRetangulo[10];

   // Quantos objetos-retângulo existem neste momento?...
```

```
    eMaisEstesRets[0].imprimeNumRetangulos();

    // Vamos guardar este número para uso futuro...
    int antNumRets = eMaisEstesRets[9].devolveNumRetangulos();

    // Vamos agora apagar a tabela dinâmica...
    delete [] eMaisEstesRets;

    // Quantos objetos-retângulo há agora?...
    variosRets[0].imprimeNumRetangulos();

    // Mas...
    cout << "Mas já foram " << antNumRets << "..." << endl;
    return 0;
}
```

Neste exemplo, além do que foi referido anteriormente, foi acrescentada uma série de métodos úteis e algumas novidades:

- Em primeiro lugar, foram adicionados mais dois construtores alternativos (por sobrecarga de função) que permitem definir implicitamente os parâmetros do retângulo e copiar os dados de um retângulo já existente. Em todos os construtores, é incrementado o membro estático `numRet`.
- Pode ser também definido um destrutor que é chamado sempre que um objeto é destruído. Supostamente, será nesse método que se faz a "limpeza da casa", como libertar a memória ocupada dinamicamente, *etc*. No nosso caso concreto, decrementa-se apenas o contador de objetos.
- O ponteiro `this` nada mais faz que conter o endereço do objeto na memória e é sempre implicitamente usado quando se faz referência a um membro a partir de um método da classe...
- A criação e destruição de ponteiros/tabelas dinâmicos/as de objetos utilizando os operadores `new` e `delete`: podem usar-se quaisquer tipos de dados com estes operadores, sejam classes ou não. Apenas se pode criar e destruir dinamicamente um objeto com um ponteiro, ou uma tabela de objetos se forem usados os parênteses retos.

6.4. Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à programação orientada a objetos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios que não for possível realizar durante as aulas práticas.

Problema 6.1 – Fácil

Acrescente métodos à classe base `CRetangulo` que permitam calcular a área e o perímetro do retângulo. Estes métodos não devem aceitar qualquer parâmetro. Implemente também um método que verifique se "este" retângulo está contido noutro retângulo passado como parâmetro por referência, ou seja se o primeiro retângulo tem uma largura e um comprimento menores ou iguais aos do segundo retângulo.

Problema 6.2 – Fácil

Implemente uma classe que permita representar a ficha de um aluno, com nome do aluno, *username* e nome do curso (e.g., MiEEC), e que armazene também as suas classificações em 9 disciplinas diferentes: PdC, ALG, AM1, LSD, AM2, LEC, EDA, MO e SMP. Implemente três métodos: (1) um que possibilite calcular e imprimir a média das suas classificações; (2) outro que devolva a nota máxima; (3) e um último que imprima o nome da disciplina a que o aluno obteve a nota mínima.

Problema 6.3 – Médio

Defina uma classe que permita representar um dispositivo elétrico. Deve permitir definir o nome/referência do dispositivo (`string` de 10 carateres), a potência de consumo, uma `struct` para definir a data de validade (no formato dd/mm/aaa) e um membro (de que tipo?) que permita definir se se encontra dentro da garantia (`true` ou `false`).

Problema 6.4 – Médio

Implemente uma classe denominada `CCash` que permite definir um preço em Euros. A classe deve armazenar um valor em dois membros inteiros diferentes: `euros` e `centimos`. Implemente um construtor por defeito (que coloca o preço em 1.99€) e um construtor por enumeração (que recebe o preço em Euros e em Cêntimos). Implemente também o método `troco()` e o método `imprime()`. O primeiro recebe como parâmetro passado por referência um objeto `CCash` com o valor pago em dinheiro e devolve um objeto `CCash` com o montante a devolver como troco, assumindo que o preço a pagar está guardado "neste objeto" (i.e. no objeto para o qual é chamado o método). O segundo mostra no ecrã o conteúdo do objeto (por ex. no formato "5,02 Eur"), formatando corretamente a sua apresentação na consola com os manipuladores da biblioteca `iomanip`.

Problema 6.5 – Médio

Dada a seguinte definição da classe `CData` que permite representar datas, implemente o construtor e os métodos indicados. Implemente também um novo método privado que permita verificar se uma dada data é ou não válida.

```
class CData {
    int dia;
```

```
    int mes;
    int ano;
    int horas;
    int minutos;
  public:
    CData();                      // Construtor por defeito (00:00 de 01.01.1970)
    CData(int d, int m, int a, int h, int min);     // Construtor por enumeração
    int devolveDia();             // Devolve o dia da data atual
    int devolveMes();             // Devolve o mês da data atual
    int devolveAno();             // Devolve o ano da data atual
    void mudaHoraDoDia(int, int);// Permite alterar a hora do dia
    void mudaData(int, int, int);// Permite introduzir uma nova data
     void escreveData();          /* Escreve hora e data no ecrã no
                                        formato hh:mm de dd/mm/aaaa */
    int numeroFDS();              /* Devolve o número de fins de semana decorridos
                                        desde o início do ano */
};
```

Problema 6.6 – Médio

Defina uma classe que permita representar frações inteiras com `numerador` e `denominador`. Implemente os três construtores e também um método que permita reduzir uma fração à sua forma irredutível (por exemplo transformar 4/8 em 1/2). Implemente também um método que permita apresentar a fração no ecrã. Preveja o caso em que o denominador seja igual a 1, passando a fração a ser representada sem denominador; por exemplo, a fração 2/1 deverá ser apresentada apenas como 2.

Problema 6.7 – Difícil

Defina uma classe que permita representar *passwords* recorrendo a alocação dinâmica de memória. Implemente um construtor por enumeração (aceita uma `string` que defina uma *password*) e por cópia. Implemente ainda os seguintes métodos:

`comprimentoPassword()` – Devolve o comprimento da *password* em número de carateres;

`baralhaPassoword()` – Troca a segunda com a primeira metade da *password* (por exemplo, "tpb3u8as" passa a "u8astpb3");

`apagaMetadePassword()` – Elimina a primeira metade da *password* formando uma nova *password* com metade do tamanho dos carateres da original. Se uma *password* contiver N carateres, a nova *password* será formada pelos N/2 últimos caracteres da original (por exemplo, "latina" passa a "ina"). Analise o que acontece a *passwords* com número ímpar de carateres;

`converteMinusculas()` – Converte a *password* para letras minúsculas.

Problema 6.8 – Médio **(adaptado do exame de recurso de 05/07/2012)**

Declare uma classe denominada `CDiametro` que permite definir um diâmetro em pés e polegadas (ambos inteiros), tendo em conta as alíneas seguintes.

a) Implemente um construtor por omissão (por defeito), que coloca o diâmetro em 10' 0" (10 pés e 0 polegadas) e um construtor por enumeração (que recebe o comprimento em pés e polegadas, sendo as polegadas facultativas, tendo o valor 0 por omissão).

b) Nesta classe, as polegadas são sempre inferiores a 12 (1 pé = 12 polegadas). Crie um método chamado `corrige_polegadas()` para garantir esta restrição, corrigindo o valor (das polegadas e pés) no caso de as polegadas serem superiores a 12 (as polegadas podem ser > 24 ou qualquer outro múltiplo de 12).

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 6.9 – Médio **(adaptado do teste de frequência de 25/03/2015)**

Considere uma classe para gerir uma tabela ordenada de números reais (e.g. amostras de temperaturas). A classe aloca dinamicamente memória para guardar a tabela. Os atributos (privados) são a tabela propriamente dita (o primeiro elemento é apontado por `tab`), a sua dimensão (`max`) e um inteiro (`n`) com o número de elementos na tabela. A tabela deve estar sempre ordenada por ordem decrescente, com os dados agrupados no início da tabela.

a) Declare a classe com os métodos (públicos) seguintes: construtor por omissão – cria uma tabela vazia com dimensão 100 (aloca memória e atribui zero a `n`; construtor com um parâmetro – cria uma tabela vazia com dimensão genérica indicada pelo parâmetro; destrutor; construtor por cópia – cria uma tabela que é uma cópia de outra recebida como parâmetro; `apaga()` – apaga (esvazia) a tabela, i.e. atribui zero a `n`; `vazia()` – verifica se a tabela está vazia; `insere()` – acrescenta um elemento ordenadamente, devolvendo `false` se a inserção falhar por a tabela estar cheia (`true` caso contrário). A tabela está cheia se `n == max`.

b) Implemente os construtores e o destrutor.

c) Implemente os restantes métodos. Na implementação do método `insere()`, utilize o princípio do algoritmo de ordenação por bolha, assumindo que a tabela está ordenada antes da inserção do novo elemento (dado como parâmetro deste método).

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 6.10 – Médio **(saiu no teste de frequência de 29/03/2017)**

A classe de objetos `CDuracao` permite definir uma duração temporal em horas e minutos. Possui os atributos `horas` e `minutos` (tipo `int`) e os métodos seguintes: i) um construtor por defeito/omissão (objeto com uma duração por defeito igual a 1h30m); ii) um construtor por enumeração (recebe dois inteiros com uma duração qualquer em horas e minutos); iii) um construtor por cópia; iv) o método `compara()` que recebe como parâmetro, por referência, outro objeto da classe `CDuracao` e devolve um n.º inteiro negativo, zero ou um n.º positivo, consoante o objeto seja menor (em duração temporal), igual ou superior ao parâmetro, respetivamente.

a) Escreva a declaração da classe de objetos `CDuracao`. Nesta alínea, não deve implementar qualquer método.
b) Implemente os construtores por defeito, por enumeração e por cópia da classe `CDuracao`.
c) Implemente o método `compara()` da classe `CDuracao`.

Implemente uma função main() para ir testando a implementação da classe.

Problema 6.11 – Médio **(saiu no teste de frequência de 13/04/2018)**

A classe de objetos `CNumComplexo` permite representar números complexos, i.e. números com parte real e parte imaginária e realizar operações com os mesmos. Possui os atributos `preal` e `pimag` (tipo `double`) e os métodos seguintes: i) construtor por defeito/omissão (número real 1+0j); ii) construtor por enumeração (recebe dois números reais com os valores da parte real e parte imaginária); iii) o método `compara()` que compara o objeto com outro objeto passado como parâmetro e devolve um booleano consoante dois números complexos sejam ou não iguais.

a) Escreva a declaração da classe de objetos `CNumComplexo`. Nesta alínea, não deve definir/implementar qualquer método.
b) Defina os construtores por defeito e por enumeração da classe `CNumComplexo`.
c) Defina o método `compara()` da classe `CNumComplexo`.

Implemente uma função main() para ir testando a implementação da classe.

Problema 6.12 – Fácil (saiu no teste de frequência de 20/03/2019)

A classe de objetos `Odometria` representa dados da odometria de um robô que se move num plano, i.e. da sua pose (posição e orientação) num dado instante de tempo. Possui os atributos `x`, `y`, e `theta` (todos números reais de dupla precisão). Possui os métodos públicos: i) construtor por defeito/omissão (todos os atributos com valor `0.0`); ii) construtor por enumeração (recebe três números reais correspondentes a `x`, `y`, e `theta`, respetivamente); iii) `void simplifOrient(void)` que converte o valor do atributo `theta` de forma a reduzir o ângulo de orientação do robô a um ângulo pertencente ao intervalo]-π; π].

a) Escreva a declaração da classe de objetos `Odometria`. Nesta alínea, não deve definir/implementar qq. método.
b) Defina os construtores por defeito e por enumeração da classe `Odometria`.
c) Defina o método `simplifOrient()`. Defina a constante `PI` (constante matemática π). O algoritmo consiste em somar ou subtrair `2*PI` ao ângulo `theta` até que este se situe no intervalo pretendido.

Implemente uma função main() para ir testando a implementação da classe.